What is a Compiler

• A compiler translates
  – from a source language S
  – to a target language T
  – while preserving the meaning of the input
Structure of a Compiler
Compiler Structure: Front End

- program text
- syntactic analyzer
- interm. rep.
- code generator
- machine code

- tokenizer
- token stream
- parser
Compiler Structure: Back End

- Program text
  - Syntactic analyzer
    - Intermediate representation
      - Code generator
        - Machine code
          - Machine independent options
            - Intermediate representation
              - Basic code generator
                - Machine dependent options
Tokenizing

- The first step in compilation
  - takes the input (a character stream) and converts it into a token stream
  - Tokens have attributes

- Technology
  - Convert regular expressions into
  - NFA and then convert into
  - DFA
Regular Expressions

• Concatenation, alternation, Kleene closure, and parenthesization

• Regular definitions
  – multiline regular expressions

• Exercise: Write regular definitions for
  – All strings of lowercase letters that contain the five vowels in order
  – All strings of lowercase letters in which the letters are in ascending lexicographic order
  – Comments, consisting of a string surrounded by /* and */ without any intervening */
**NFAs**

- Finite collection of states
- Edges are labelled with letters
- One start state
- (Wlog) one accepting state
- **Exercise:** Convert these to NFA
  - $a|b$
  - $(a|b)c$
  - $(a|b)^*c$
  - $(a|b)^* a (a|b)(a|b)$
**DFAs**

- Like NFAs, but
  - all the outgoing edges of any node have distinct labels
- Any NFA can be converted to an equivalent DFA
- **Exercises:**
  - Convert to DFA:
**Parsing**

- **Purpose**
  - Convert a token stream into a parse tree

\[
x = x + 1
\]

\[
<\text{id, "x"}> <\text{assign}> <\text{id, "x"}> <\text{plus}> <\text{number, "1"}>
\]
Context-Free Grammars

• Context-free grammars have
  – terminals (tokens)
  – non-terminals
  – sentential forms
  – sentences

• Derivations
  – Derive \( id + id \ast id \) with this grammar:

\[
E \rightarrow E + T \mid T \\
T \rightarrow T \ast F \mid F \\
F \rightarrow ( E ) \mid id
\]
Derivations

• Leftmost (rightmost) derivations
  – Always expand the leftmost (rightmost) non-terminal

• Derivations and parse trees
  – Internal nodes correspond to non-terminals
  – Leaves correspond to terminal

• Ambiguity
  – When a string has more than one derivation
  – Can result in different parse trees
Derivations and Parse Trees

\[
\begin{align*}
E & \rightarrow E + E \\
E & \rightarrow E + \text{id} \\
E & \rightarrow E \ast E + \text{id} \\
E & \rightarrow E \ast \text{id} + \text{id} \\
( E ) & \rightarrow ( E ) \ast \text{id} + \text{id} \\
( E + E ) & \rightarrow ( E + E ) \ast \text{id} + \text{id} \\
( \text{id} + E ) & \rightarrow ( \text{id} + E ) \ast \text{id} + \text{id} \\
( \text{id} + \text{id} ) & \rightarrow ( \text{id} + \text{id} ) \ast \text{id} + \text{id}
\end{align*}
\]
Left-Recursion

• Left-recursion makes parsing difficult

• Immediate left recursion:
  – \( A \rightarrow A\alpha | \beta \)
  – Rewrite as: \( A \rightarrow \beta A' \) and \( A' \rightarrow \alpha A' | \varepsilon \)

• More complicated left recursion
  – \( A \rightarrow^+ A\alpha \)
**Left Factoring**

- Makes a grammar suitable for top-down parsing
- For each non-terminal $A$ find the longest prefix $\alpha$ common to two or more alternatives
  - Replace $A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \alpha \beta_3 | ... | \alpha \beta_n$ with
  - $A \rightarrow \alpha A'$ and $A' \rightarrow \beta_1 | \beta_2 | \beta_3 | ... | \beta_n$
- Repeat until not two alternatives have a common prefix
Exercise

- **Exercise:**
  - Remove left recursion
  - Left-factor

\[
\begin{align*}
\text{rexpr} & \rightarrow \text{rexpr} + \text{rterm} \mid \text{rterm} \\
\text{rterm} & \rightarrow \text{rterm} \text{ rfactor} \mid \text{rfactor} \\
\text{rfactor} & \rightarrow \text{rfactor} \ast \mid \text{rprimary} \\
\text{rprimary} & \rightarrow a \mid b
\end{align*}
\]
**First and Follow**

- **FIRST(X)** : The set of terminals that begin strings that can be derived from $X$
- **FOLLOW(X)**: The set of terminals that can appear immediately to the right of $X$ in some sentential form

- **Be able to:**
  - compute FIRST and FOLLOW for a small example
**FIRST and FOLLOW Example**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow T E'$</td>
<td></td>
</tr>
<tr>
<td>$E' \rightarrow + T E' \mid \varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$T \rightarrow F T'$</td>
<td></td>
</tr>
<tr>
<td>$T' \rightarrow * F T' \mid \varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$F \rightarrow (E) \mid \text{id}$</td>
<td></td>
</tr>
</tbody>
</table>

- FIRST($F$) = FIRST($T$) = FIRST($E$) = {$(, \text{id}$
- FIRST($E'$) = {$+, \varepsilon$}
- FIRST($T'$) = {$*, \varepsilon$}
- FOLLOW($E$) = FOLLOW($E'$) = {$(, \text{$}$
- FOLLOW($T$) = FOLLOW($T'$) = {$+, \text{)}$, \text{$}$
- FOLLOW($F$) = {$+, *, \text{)}, \text{$}$
**LL(1) Grammars**

- Left to right parsers producing a leftmost derivation looking ahead by at most 1 symbol

- Grammar $G$ is LL(1) iff for every two productions of the form $A \rightarrow \alpha | \beta$
  - $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$ are disjoint
  - If $\varepsilon$ is in $\text{FIRST}(\beta)$ then $\text{FIRST}(\alpha)$ and $\text{FOLLOW}(A)$ are disjoint (and vice versa)
**LL(1) Parser**

- LL(1) Parsers are driven by a table
  - Non-terminal x Next token => Expansion

- **Be able to:**
  - fill in a table given the FIRST and FOLLOW sets
  - use a table to parse an input string
Bottom-up Parsing

- Shift-reduce parsing
  - Won't be covered on the exam
Type-Checking

• Type checking is done by a bottom-up traversal of the parse tree
  – For each type of node, define what type it evaluates to given the types of its children
  – Some extra types may be introduced
    • error type
    • unknown type
  – These can be used for error recovery

• Environments
  – Used for keeping track of types of variables
  – Static lexical scoping

• Exercise:
  – pick a parse tree and assign types to its nodes
Parse DAGs

• Parse DAGS
  – Like parse trees
  – Common subexpressions get merged

• Exercise:
  – Construct the parse DAG for
    • \((x+y)-((x+y)*(x-y))\)
    • \(((x1-x2)*(x1-x2))+((y1-y2)*(y1-y2))\)
Intermediate Code Generation

- Two kinds of intermediate code
  - Stack machine
  - 3 address instructions
- For 3AI
  - Assign temporary variables to internal nodes of parse dags
  - Output the instructions in reverse topological order
- For stack machines
  - Just like in assignment 3
- Recipes for control structures
Example

- **Exercise**: Generate 3AI and stack-machine code for this parse tree
Scope and Code Generation

• The interaction between static lexical scope and the machine stack
  – Frame pointers
  – Parent frame pointers
  – The frame pointer array

• Object scope
  – Inheritance
  – Virtual methods
    • dispatch tables

• Be able to:
  – Illustrate state of stack fp, and fpp for a function call
  – Illustrate memory-layout of an OOP-language object
Basic Blocks

• Blocks of code that always execute from beginning to end

• **Be able to:**
  – Given a program, compute the basic blocks

• Next-use information:
  – Lazy algorithm for code generation and register usage based on next-use information

• **Be able to:**
  – Compute next-use information for all the variables in a basic block
  – Illustrate a register allocation based on next-use information

• The dangers of pointers and arrays
Basic Blocks as DAGS

• Applications
  – Dead-code elimination, algebraic identities, associativity, etc

• Be able to:
  – Given a basic block, compute its DAG representation
  – Reassemble a basic block from its DAG
    • be careful with pointers and arrays
Peephole Optimization

- Different kinds of peephole optimizations
  - redundant load/stores
  - unreachable code
  - flow of control optimizations (shortcuts)
  - algebraic simplifications and reduction in strength
The Control Flow Graph

- Indicates which basic blocks may succeed other basic blocks during execution

- **Be able to:**
  - Compute a control flow graph
  - Choose register variables based on the control-flow graph
  - Eliminate unreachable code
  - Find no-longer-used variables
  - Compute the transitive closure
Register Allocation by Graph Coloring

• The interference graph
  – nodes are variables
  – two nodes are adjacent if the variables are active simultaneously
  – Color the graph with the minimum number of colors

• Inductive graph coloring algorithm
  – Delete vertex of lowest degree
  – Recurse
  – Reinsert vertex and color with lowest available color

• Be able to:
  – Illustrate inductive coloring algorithm
Ershov Numbers

• Computed by bottom-up traversal of parse tree
• Represent the minimum number of registers required to avoid loads and stores
• Dynamic programming extension
  – reorderings of children
  – different instructions

• Be able to:
  – Compute Ershov numbers
  – Compute dynamic programming costs
Data-Flow Analysis

• Define in and out
  – for each line of code
  – for each basic block

• Define transfer functions
  – out[L] = f(L, in[L])
  – in[B] = f(out[B1],...,out[Bk])
    • where B1,...,Bk are predecessors of B
  – Sometimes works backwards

• Example applications
  – reaching definitions, undefined variables, live variable analysis

• Be able to:
  – Apply iterative algorithm for solving equations
The GNU Compiler Collection

• History and background
  – Started in 1985
  – Open source
  – Compilation steps:
    • Input language
    • Parse tree
    • GENERIC
    • GIMPLE
    • RTL
    • Machine language

• Be able to:
  – Recognize a picture of Richard Stallman
  – Know difference between GENERIC and GIMPLE
Want to Build a Compiler

- Cross-compiling
- Bootstrapping
- Self compiling
- T-diagrams
- **Be able to:**
  - Understand T-diagrams
  - Solve a cross-compilation problem
LL(1) Parser
Want to Write a Compiler?

- A compiler has 3 main parameters
  - Source language (S)
    - What kind of input does the compiler take?
      - C, C++, Java, Python, ...
  - Implementation language (I)
    - What language is the compiler written in?
      - C, Java, i386, x86_64
  - Target language (T)
    - What is the compiler's target language
      - i386, x86_64, PPC, MIPS, ...
Source Language Issues

• Complexity
  – Is a completely handwritten compiler feasible?

• Stability
  – Is the language definition still changing?

• Novelty
  – Do there already exist compilers for this language?

• Complicated, or still-changing languages promote the use of compiler generation tools
**Target Language Issues**

- **Novelty**
  - Is this a new architecture?
  - Are there similar architectures/instruction sets?

- **Available tools**
  - Is there an assembler for this language?
  - Are there other compilers for this language?
Performance criteria

• Speed
  – Does it have to be a fast compiler?
  – Does it have to be a small compiler?
  – Does it have to generate fast code?

• Portability
  – Should the compiler run on many different architectures (rehostability)
  – Should the compiler generate code for many different architectures (retargetability)
Possible Workarounds

• Rewrite an existing front end
  – when the source is new
  – reuse back (code generation) end of the compiler

• Rewrite an existing back end
  – when the target architecture is new
  – retarget an existing compiler to a new architecture

• What happens when both the source language and target language are new?
  – Write a compiler from scratch?
  – Do we have other options?
Composing Compilers

- Compilers can be composed and used to compile each other

- Example:
  - We have written a Java to JVM compiler in C and we want to make it to run on two different platforms i386 and x86_64
  - both platforms have C compilers
Example

- Assignment 3:
- Assignment 4:
Example

- Show how to
  - To take your PRM compiler and make it faster
  - To take your Jasmin optimizer and make it faster
Bootstrapping by cross-compiling

• Sometimes the source and implementation language are the same
  – E.g. A C compiler written in C

• In this case, cross compiling can be useful
Bootstrapping Cont'd

- Bootstrapping by reduced functionality
  - Implement, in machine language, a simplified compiler
    - A subset of the target language
    - No optimizations
  - Write a compiler for the full language in the reduced language
**Bootstrapping for Self-Improvement**

- If we are writing a good optimizing compiler with $I=S$ then
  - We can compile the compiler with itself
  - We get a fast compiler

- gcc does this (several times)
Summary

• When writing a compiler there are several techniques we can use to leverage existing technology
  – Reusing front-ends or back ends
  – Cross-compiling
  – Starting from reduced instruction sets
  – Self-compiling